# Toward a Query Language on Simulation Mesh Data: an Object-oriented Approach

Byung S. Lee, Robert R. Snapp
Department of Computer Science
University of Vermont
Burlington, Vermont, U.S.A.
{bslee, snap}@cs.uvm.edu

Ron Musick
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, California, U.S.A.
(Currently with Ikuni, Inc.) musick@ikuni.com

## Abstract

*As simulation is gaining popularity as inexpensive means of experimentation in diverse fields of industry and government, the attention to the data generated by scientific simulation is also increasing. Scientific simulation generates mesh data, i.e., data configured in a grid structure, in a sequence of time steps. Its model is complex – understanding it involves mathematical topology and geometry in addition to fields (in the relational sense). Moreover, there is no query language developed on mesh data at all. These are what we address in this paper. We develop a comprehensive model of mesh data in an object-oriented manner, propose a set of primitive algebraic operators, show their object-oriented implementation, and demonstrate that the well-known object query language ODMG OQL is powerful enough to express queries on mesh data, whether the queries are on mesh topology, geometry, fields, or combination of them. Finally, we discuss some physical implementation issues pertinent to executing queries efficiently.*

## 1 Introduction

Simulation is a cost-effective way of conducting a test without actually building a product, and has been in use in various fields including physical science, defense experiment, engineering design, medical imaging, oil exploration, and weather forecasting. Many scientific and engineering simulation data are produced in a mesh data structure, such as NetCDF [1], HDF [2], and SILO [3]. Scientists are producing simulation data in the size of tera-byte order, but have very limited tools available for exploring and querying the produced data. There are visualization tools [4, 5] available today, which provide a few fixed forms of primitive query operations such as finding points, displaying iso-surfaces, or displaying orthogonal slices. However, users need a more powerful query facility that will enable them to interactively search the simulated mesh data in an ad hoc manner. There has been some similar work done in other context to address declarative, ad hoc query languages [7, 8]. However, there is none well accepted as a mesh data query language in the scientific computing community yet.

The purpose of this paper is to establish an object-oriented framework for queries on mesh data, and propose its data model and query language. More specifically, we demonstrate that a model of mesh data can be built using object-oriented data structure and operations, thus we can naturally choose to use ODMG (Object Data Management Group) OQL [6] as the query language. In addition, we categorize queries on mesh data into topology queries, geometry queries, and field queries, and present examples of probable queries for each category. From the operations defined in these three categories, we propose a *base set* of algebraic operators on mesh data and the associated notion of *completeness*, and demonstrate that other operators can be derived from those base set operators either algebraically or computationally. Lastly, because a query language is not useful unless it can be executed efficiently, we discuss our considerations for improving the efficiency of executing queries on mesh data.

We claim the following points as our contributions made through this paper. First, we developed an object-oriented model of mesh data, and showed an implementation of mesh data structure and operations on it as C++ classes. Secondly, we demonstrated that OQL was sufficient to express all probable queries on mesh data, where each probable query exemplifies operations on mesh data. Thirdly, we discussed efficiency considerations in implementing the query language, such as adding collection extents, clustering, and indexing.

This paper is organized as follows. Following this introduction, we first give an overview of simulation mesh data in Section 2, and describe its object-oriented model in Section 3. Then in Section 4 we demonstrate that OQL is adequate enough to express queries on mesh data, and present our query implementation considerations in Section 5. Finally, summary and further work follow in Section 6.

## 2 Overview of Simulation Mesh Data

In this section, we give a concise description of the concepts and characteristics of simulation mesh data, and illustrate it with an example. We can understand mesh data as a discrete representation of continuous data, exhibiting a grid structure. Scientific and engineering simulation typically generates mesh data in a sequence of time steps as the result of solving partial differential equations. As mentioned in Introduction, the application of simulation mesh data is numerous, predominantly in science and engineering. Simulation mesh data can be categorized based on the following characteristics. (See [9] for a more comprehensive discussion.)

*Regularity* – Mesh data may be regular or irregular depending on the geometric pattern of the points for which values are computed. We can distinguish between spatial regularity and temporal regularity. Spatially regular mesh data has data values computed at “a regular grid or some other geometric structure” [9]. Temporally regular mesh data has data values computed at regular time intervals. Irregular mesh data requires that the spatial or temporal coordinates must be stored together with the computed values. In contrast, regular mesh data allows the coordinates to be calculated, hence not stored.

*Time-variation* – Mesh data may be time-invariant or time-varying depending on the variation of the geometric coordinates of points over time. Time-varying mesh is very common in simulations of dynamic processes, such as deforming an artifact or changing natural phenomenon. Examples are simulating a car crash, bending a rod, or weather change. Unlike time-invariant mesh data, time-varying mesh data requires that the neighborhood relationships (i.e., topology) of mesh grid points must be stored in addition to the coordinates (i.e., geometry) of points so that points whose coordinates change over time can be traced at different time points.

*Density* – Mesh data may be dense to a varying degree depending on how many mesh points are empty, that is, have no data associated with them. For example, a simulation of air turbulence would have field values (e.g., velocity, orientation) at every point, therefore dense. By contrast, simulation of particle collision would have field values at only a small number of points, therefore sparse. [9]

Figure 1 shows an example of mesh data generated by simulation of a can being crushed against a wall. It shows a snapshot taken at the 8th time step in a sequence of 44 simulation steps 0 through 43. The crushing can data is stored in a four dimensional Euclidean space defined by spatial variables $x$, $y$, $z$ and time step. Its geometrical structure at each time step is lying on a cylindrical coordinate system, and its topological structure is a hexahedral grid. There are

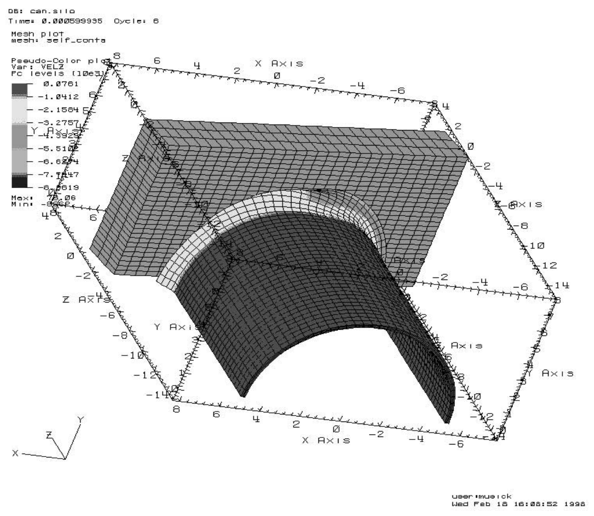

**Figure 1. Crushing can mesh data**

ten field variables – displx, disply, displz, velx, vely, velz, accx, accy, accz, and eqps. “Displ”, “vel”, and “eqps” denote “displacement”, “velocity”, and “acceleration”, respectively. Each has components in x, y, and z directions. The last field variable “eqps” denotes “equivalent plastic strain”, which is a measurement of strain or stress on the can surface. All fields except “eqps” are vertex-centered (i.e., defined at a mesh vertex) whereas “eqps” is zone-centered. In this three dimensional space defined at each time step, a zone corresponds to a volume. A zone corresponds to a face in a two dimensional space. The value of a zone-centered field is calculated by interpolating the values of the fields at adjacent vertices.

## 3 Object-oriented Model of Simulation Mesh Data

A rigorous model of mesh data requires sound mathematical and physical theories (such as “sheaf data model” [10] based on fiber bundles [11]), and involves formal models of mesh fields, mesh topology, and mesh geometry. In this section, we present a rather informal model that adequately supports our object-oriented query language. In other words, we present a model that includes what we deem a complete set of primitive operators on mesh fields, mesh topology, and mesh geometry without necessarily relying on a formal theory as in [10]. More specifically, we identify probable query types for each of the three operation categories, and propose a data structure and operators on the data structure. Empirically we observed that the operators are sufficient to support any query on mesh data that can be answered using algorithmic computation. We also present probable queries that combine two or three categories of the operations. In addition, we propose a base set of operations with which we can retrieve any information from mesh that does not require programming.

## 3.1 Topological operations

Consider the following probable query types that use operations on mesh topology:

**T-A** Find all mesh elements on the boundary of mesh.

**T-B** Find all mesh elements that are bounded by other mesh elements (at a lower level). – This is called an "adjacency query."

**T-C** Find all mesh elements of certain topological characteristics (e.g., tetrahedral volumes).

where the mesh elements mentioned above are one of vertices, edges, faces, and volumes. We propose a topological data structure based on the data structure used in computer graphics [12], as well as operations that can be used to express all these queries, as follows. The data structure is a *mesh topology graph* $\mathcal{G}$ as shown in Figure 2. If a mesh consists of two or more submeshes, $\mathcal{G}$ is defined as $\{\mathcal{G}_i | i = 1, 2, \cdots\}$ where each $\mathcal{G}_i$ is the topology graph of a submesh. The topology graph $\mathcal{G}_i$ of a submesh is defined as $\mathcal{G}_i = \{< E_i, L_i >\}$ where $E_i$ denotes a set of submesh elements, and $L_i$ denotes a set of parent-child links between the members of $E_i$. Each level $l$ of the graph contains a set $E_i^l$ of homogeneous elements. For example, for a sequence of meshes generated in a 3-dimensional space, level 0 contains mesh vertices, level 1 contains edges, level 2 faces, level 3 volumes, and level 4 hyper-volumes. In Figure 2 a hyper-volume is shown as a time sequence of volumes. Generically, we call a mesh element at level $l$ an *$l$-dimensional volume*. A mesh element at level $l$ has as its components mesh elements at level $l-1$, for $l = 1, 2, 3, 4$. Conversely, a mesh element at level $l$ is a component of one or more mesh elements at level $l+1$ for $l = 0, 1, 2, 3$. We define a *$k$-dimensional mesh* as a mesh whose mesh element at the highest level is a $k+1$-dimensional volume, which is a time sequence of $k$-dimensional volumes. In theory, the maximum level of dimension can be an arbitrary positive integer.

As to the operators, let $E$ denote a set of mesh elements, and let parent($e$, $e'$) denote a predicate "$e$ is a parent of $e'$", then, we define the following topological operators. (The operators marked "derived" can be derived from the ones marked "primitive". Due to the limit on space, we omit the verification of the derivability.

- $e$.Parents() $= \{e' \in E | \text{parent}(e', e)\}$ [primitive]
- $e$.Children() $= \{e' \in E | \text{parent}(e, e')\}$ [primitive]
- $e$.Siblings() $= \{e' \in E | \exists p (e' \neq e \wedge \text{parent}(p, e) \wedge \text{parent}(p, e'))\}$ [derived]

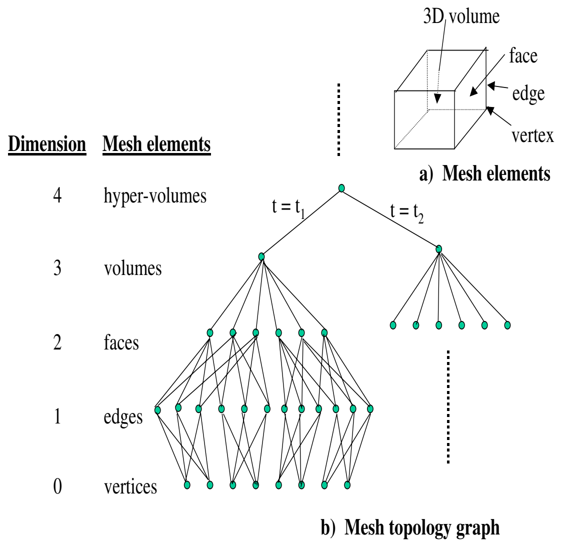

**Figure 2. Mesh topology graph of a 3-dimensional mesh**

- $e$.Mates() $= \{e' \in E | \exists c (e' \neq e \wedge \text{parent}(e, c) \wedge \text{parent}(e', c))\}$ [derived]

For example, edg.Parents() returns the faces that are delimited by an edge edg; edg.Children() returns the vertices that delimit the edge edg; edg.Siblings() returns the edges that delimit the same face; and edg.Mates() returns the edges that share the same vertex that delimits them.

## 3.2 Geometrical operations

Probable query types that need operations on mesh geometry are:

**G-A** Find the coordinates of mesh vertices.

**G-B** Find the mesh vertex closest to a given point.

Note that geometrical operations are applied to mesh vertices only. We propose a *vertex coordinate set* as the data structure, which is defined as $\mathcal{N} = \{< n, \vec{X} >\}$ where $n$ denotes a vertex and $\vec{X}$ denotes a vector representation of the coordinate $X =< x_0, x_1, \cdots, x_k >$ of a vertex $n$ in a $k$-dimensional mesh. For example, for a 4-dimensional mesh, $\vec{X}$ = vector($t, x, y, z$) where $t$ denotes a simulation time step. As to the operators, we propose the following ones:

- $n$.Coordinate() returns the vector coordinate of a mesh vertex $n$. [primitive]
- Given a set $N$ of mesh vertices, $N$.NearestVertex(&$m$(vector<double>&,

vector<double>&), $\vec{X}_c$) returns a vertex $n$ closest to a reference point at coordinate $X_c$ among all elements of $N$, where the distance between $X_c$ and the coordinate $X_j$ of a vertex $n_j \in N$ is calculated using a distance metric function $m$ as $m(\vec{X}_c, \vec{X}_j)$. Note that $\vec{X}_j = n_j$.Coordinate(). [derived]

An example of NearestVertex is NearestVertex(&sumsq( vector<double>&, vector<double>&), vector(30, 2,3, 1.5, 7.6)) where the second argument is the vector representation of a reference point coordinate, and sumsq ("sum of squares") is a distance metric between the reference point coordinate and the coordinate $X_j$ of a vertex $n_j$. In this case, for a given vertex at coordinate $X_j =< t_j, x_j, y_j, z_j >$, its distance to the reference point is calculated as sqsum($\vec{X}_c, \vec{X}_j$) = $(t_j - 30)^2 + (x_j - 2.3)^2 + (y_j - 1.5)^2 + (z_j - 7.6)^2$.

## 3.3 Field operations

Probable query types that need operations on mesh field variables are:

**F-A** Find the values of field variables of a mesh element.

**F-B** Find the set of mesh elements that satisfy a predicate on the values of field variables.

where mesh elements are one of mesh vertices, edges, faces, and volumes. For a data structure we propose a *feature vector set* defined as $\mathcal{F} = \{< e, \vec{F} >\}$ where $e$ denotes a mesh element and $\vec{F}$ denotes a vector representation of field variables of the mesh element $e$. The proposed operators are:

- $e$.Fields() returns the values of a field vector $\vec{F}$ defined at a mesh element $e$. This operation is analogous to the relational project operator, which can be written as $\Pi_{\vec{F}} e$. [primitive]

- Given a set $E$ of mesh elements, $E$.Elements($P(\vec{F})$) returns a subset of $E$ whose field vector $\vec{F}$ satisfies a predicate condition $P$ on it. This operation is analogous to the relational select operator, which can be written as $\sigma_{P(\vec{F})} E$. [derived]

For example, given a set $E$ of mesh elements at a particular level, $E$.Elements("temp > 13") returns a set of mesh elements whose field variable temp is greater than 13.

## 3.4 Combined operations on mesh data structure

Consolidated from the data structures for the three categories of mesh operations, the entire data structure of each disjoint mesh is defined as an ordered triplet $< \mathcal{G}, \mathcal{N}, \mathcal{F} >$ where $\mathcal{G}$ is a mesh topology graph, $\mathcal{N}$ is a vertex coordinate set, and $\mathcal{F}$ is a feature vector set. We give here some examples of query types that use a combination of two or more categories of mesh operations. (In the following examples, the symbols T, G, F in square brackets denote topological, geometrical, and field operations, respectively, that are used by the corresponding query.)

**C-A** Find the coordinates of mesh vertices delimiting a mesh element. [TG]

**C-B** Find mesh volumes containing a point. [TG]

**C-C** Find the values of field variables of all mesh vertices in a range of spatial coordinates. [GF]

**C-D** Find all mesh elements on the contact interface between two meshes. [FT]

**C-E** Compare the value of a field variable at a mesh element and an aggregate value of a field variable at neighboring mesh elements. [FT]

**C-F** Find an aggregate value of a geometric quantity at mesh elements whose field variables satisfy a predicate condition. [TGF]

## 3.5 Base set of mesh operations

We propose five operations {Dimension, Parents, Children, Fields, Coordinate} as the base set ($\mathcal{B}$) of algebraic operations on mesh data $\{< \mathcal{G}, \mathcal{N}, \mathcal{F} >\}$. A possible implementation of the mesh data structure and the base set of operations is shown below in C++ syntax.

```
/* Forward declarations */
class Element;
class Vertex;
/* Data structures */
typedef pair<Element*, Element*> Link
pair<list<Element*>, list<Link>> mtg;
typedef vector<double> X
set<pair<Vertex*, X>> vcs;
typedef vector<string> F
set<pair<Element*, F>> fvs;
typedef vector<mtg, vcs, fvs> Submesh
Submesh smesh;
set<Submesh> mesh;
/* Operations */
class Element {
  short dimension;
  vector<double> fields;
  list<Element*> parents;
  list<Element*> children;
public:
  short Dimension();
  vector<double> Fields();
  list<Element*> Parents();
  list<Element*> Children();
}
class Vertex: Element {
  vector<double> coordinate;
public:
  vector<double> Coordinate();
}
```

One interesting issue here is to assess the "completeness" of our data model denoted by the quadruple $\{<\mathcal{G}, \mathcal{N}, \mathcal{F}, \mathcal{B}>\}$. We use structural duplicability as the completeness criterion, defined as follows:

**Definition 3.1 (Completeness of a data model)** *A data model $<\mathcal{S}, \mathcal{O}>$ defined by its data structure $\mathcal{S}$ and a set of algebraic operators $\mathcal{O}$ is said to be complete if and only if $\mathcal{S}$ and $\mathcal{O}$ are necessary and sufficient to duplicate the data. In this case, we also say $\mathcal{O}$ is algebraically complete in terms of the structural duplicability of $\mathcal{S}$.*

The following assertion is self-evident:

**Axiom 3.1 (Duplicability of simulation mesh data)** *Simulation mesh data is completely duplicable given the mesh data structure $<\mathcal{G}, \mathcal{N}, \mathcal{F}>$ and the base set operators $\mathcal{B}$.*

Hence, our data model denoted by the quadruple $<\mathcal{G}, \mathcal{N}, \mathcal{F}, \mathcal{B}>$ is complete.

Although algebraically complete in terms of structural duplicability, the base set of operators alone is not enough to retrieve arbitrary information out of mesh data. This is more evident if the computation requires control flow of operations, such as conditional branching (i.e., if−then−else) or iteration (e.g., for loop). For instance, we need a more expressive language in order to implement operations marked "derived" in Sections 3.1 through 3.3. Thus, we augment the base set operators with computational operators available from a programming language (e.g., C++, Java) or an extended database language (e.g., PL/SQL, Transact-SQL). We do not show the implementation of the derived operators here due to the limit on space.

## 4 Querying Simulation Mesh Data

In this section, we show examples of creating the data structure presented in Section 3 and write in OQL the queries presented in the same section. We found OQL sufficient thanks to it extensibility of incorporating user-defined methods in a query statement.

### 4.1 Creating mesh data structure

The three structural components of mesh data – topology graph, vertex coordinate set, and feature vector set – can be implemented as abstract data types, whose schema are shown in ODMG Object Definition Language (ODL) syntax below. This schema is an ODMG ODL version of the structure shown in C++ in Section 3.5. (In Section 5, we will consider adding classes 3DVolume, Face, and Edge, and their associated extents for an efficiency reason.)

```
class Element (extent elements) {
  attribute short dimension;
  attribute list<double> fields;
  relationship list<Element> parents
    inverse Element::children;
  relationship list<Element> children
    inverse Element::parents;
  short Dimension();
  list<double> Fields();
  list<Element> Parents();
  list<Element> Children();
};
class Vertex extends Element
  (extent vertices) {
  attribute list<double> coordinate;  // <t, x, y, z>
  list<double> Coordinate();
};
```

### 4.2 Querying mesh data structure

In this section we will first explain one intuitive way of handling time steps in simulation mesh data and, based on that, demonstrate the suitability of OQL as the query language on simulation mesh data. For this, we use examples for each of the probable queries listed in Sections 3.1 through 3.4.

#### 4.2.1 Handling time steps

In a $d$-dimensional space defined by a Cartesian coordinate system $\mathbf{I_t} \times \mathbf{R_{x_1}} \times \mathbf{R_{x_2}} \cdots \times \mathbf{R_{x_d}}$, time step is denoted as the first component of the coordinate of a mesh element. Thus, what we recognize as a sequence of $d$-dimensional volumes across time steps is in essence one $d$+1-dimensional volume when we take the time step into the coordinate system. Conversely, a $d$-dimensional volume is decomposed into a series of $d$-1-dimensional volumes grouped by their time steps. In other words, all $d$-dimensional mesh volumes that share the same time step belong to the same time instance group. Note that we can check the time step of any mesh element x by applying firstChild(x) repeatedly until we reach a vertex, as shown below. For example, if v is a 3-dimensional mesh volume, we check the time step of v by checking the time step of v's first face's first edge's first vertex.

```
define firstChild(x) first(x.Children())

define timeStep(n) n.Coordinate()[0:0]
define timeStep(e)
  firstChild(e).Coordinate()[0:0]
define timeStep(f) firstChild(firstChild(f)).
  Coordinate()[0:0]
define timeStep(v) firstChild(firstChild(
  firstChild(v))).Coordinate()[0:0]
```

TimeStep function invokes the children operator, which is a topological operator. However, we exclude timeStep from consideration when classifying a query category. In fact, using a composite object index (that will be discussed in Section 5) will obviate the need for any topological navigation that follows a path from a mesh element down to a mesh vertex.

#### 4.2.2 Examples of queries on topology only

In Section 3.1, we proposed three types of probable queries on mesh topology. We show the corresponding examples here.

**T-A**: *Given a 3-dimensional mesh, find all mesh faces on the boundary of the mesh at time step 13.* (That is, faces that belong to only one volume.)

```
select f from elements f
 where f.Dimension() = 2 and timeStep(f) = 13
   and count(f.Parents()) = 1;
```

**T-B**: *Given a 3-dimensional mesh, find all mesh volumes bounded by vertices Vtx1, Vtx2, Vtx3, Vtx4, Vtx5, Vtx6, Vtx7, and Vtx8 at each time step 3 to 5.* (Note that vertices are two levels below volumes in a mesh topology graph.)

```
select v from elements  v
 where v.Dimension() = 3 and timeStep(v) >= 3
  and timeStep(v) <= 5 and set(Vtx1, Vtx2, Vtx3,
      Vtx4, Vtx5, Vtx6, Vtx7, Vtx8) <=
      distinct(select n from v.Children() f,
               f.Children() e, e.Children() n)
  group by timeStep(v);
```

**T-C**: *Given a 3-dimensional mesh, find the number of all tetrahedral volumes at time step 13.*

```
select count(v) from elements v
 where v.Dimension() = 3 and timeStep(v) = 13
   and count(v.Children()) = 4;
```

### 4.2.3 Examples of queries on geometry only

In Section 3.2, we proposed two types of probable queries on mesh geometry. Corresponding examples follow here. Note that the mesh vertex Vtx is defined at a particular time step.

**G-A**: *Given an arbitrary dimensional mesh, find the coordinate of a mesh vertex Vtx.*

```
Vtx.Coordinate();
```

**G-B**: *Given a 3-dimensional mesh, find the vertex closest to a given point at coordinate < 10.2, 15.3, 11.7 > at time step 18 using a Euclidean distance metric in a 4-dimensional space* $\mathbf{I_t} \times \mathbf{R_x} \times \mathbf{R_y} \times \mathbf{R_z}$.

```
vertices.NearestVertex(&Euclidean(vector<double>&,
vector<double>&), vector(18, 10.2, 15.3, 11.7));
```

where vertices is the extent of class Vertex, "vector" is a computational operator that creates a vector given its elements, and NearestVertex is a computationally derived operator whose definition can be readily written using the base set operator Coordinate. Due to limited space, we omit the code in this paper.

### 4.2.4 Examples of queries on fields only

In this section, we show two types of queries on mesh fields only, shown in Section 3.3.

**F-A**: *Given an arbitrary dimensional mesh, find the values of field variables "temperature" and "pressure" of a mesh vertex Vtx.* Assume we know that "temperature" and "pressure" are the 3rd and 4th field variables of a mesh vertex, respectively. Then, the query is written as follows. (Note that the vertex Vtx is defined at a particular time step.)

```
Vtx.Fields()[3:4];
```

**F-B**: *Given an arbitrary dimensional mesh, find the set of mesh vertices whose field variable "temperature" > 300.0 and field variable "pressure" > 500.0 at time step 19.* Given the extent "vertices" of class Vertex, we can write the query using the field operator Elements as follows:

```
vertices.Elements(''temperature > 300.0 and
                  pressure > 500.0'') intersect
(select n from vertices n where timeStep(n)=19);
```

Obviously, the query is inefficient. If we know the order of the field variables, we can write the same query using the Fields operator as follows, assuming that "temperature" and "pressure" are the 3rd and 4th field variables of a mesh vertex, respectively.

```
select n from vertices n
 where timeStep(n)=19 and n.Fields()[3:3] > 300.0
   and n.Fields()[4:4] > 500.0;
```

### 4.2.5 Examples of queries on topology and geometry combined

In Section 3.4, two query types require a combination of topology and geometry to answer it.

**C-A**: *Given a 3-dimensional mesh, find all vertex coordinates of a mesh volume Vol.* Note that the mesh volume Vol is defined at a particular time step.

```
select distinct n.Coordinate() from
Vol.Children() f, f.Children() e, e.Children() n;
```

**C-B**: *Given a 3-dimensional mesh, find all mesh volumes containing a point at coordinate < x = 13.0, y = 25.3, z = 17.2> at time step 15.*At each time step, there exists only one such a mesh volume unless the coordinate of the given point coincides with a vertex coordinate, which hardly happens in floating-point calculations. Thus, the query can be written as:

```
select v from elements v, v.Children() f,
f.Children() e, e.Children() n, n.Coordinate() c
 where v.Dimension() = 3 and timeStep(n) = 15
   and v.contains(13.0, 25.3, 17.2);
```

In this query, we introduced a new operator "contains." When applied to a 3-dimensional volume with a given 3-dimensional coordinate $X = < x, y, z >$, "contains" returns a Boolean constant TRUE if and only if the coordinate is within the volume, and FALSE otherwise. Implementing the operator "contains" is feasible with a computational geometry approach. For example, if we consider an imaginary point $p_o$ outside of the mesh boundary, a point $p_i$ is inside a volume if and only if any straight line connecting the two points $p_i$ and $p_o$ crosses an odd number of faces.

### 4.2.6 Examples of queries on geometry and fields combined

There is one type of query in Section 3.4 that is on a combination of mesh geometry and fields.

**C-C**: *Given an arbitrary dimensional mesh, find the values of field variables of all mesh vertices in a spatial range of 100.0 < x < 200.0, 200.0 < y < 400.0, and 50.0 < z < 70.0 in the time steps 15 through 20.* Note that c[0:0] in the following query can be replaced by timeStep(n).

```
select n.Fields()
  from vertices n, n.Coordinate() c
 where c[0:0] >= 15 and c[0:0] <= 20
   and c[1:1] > 100.0 and c[1:1] < 200.0
   and c[2:2] > 200.0 and c[2:2] < 400.0
   and c[3:3] > 50.0 and c[3:3] < 70.0;
```

### 4.2.7 Examples of queries on fields and topology combined

In Section 3.4, there are two types of queries that are on both fields and topology.

**C-D**: *Given two 3-dimensional meshes, find all mesh faces on the contact interface between the two meshes at time step 7.* Assume their materials distinguish the two mesh data. Then, we have only to check if there are faces that belong to two volumes with two distinct values of field "material" (e.g., glass and water). Assume "material" is the 4th field of a mesh volume.

```
select f from elements f, f.Parents() v
 where f.Dimension() = 2 and timeStep(f) = 7
   and count(distinct v.Fields()[4:4]) = 2;
```

**C-E**: *Given an arbitrary dimensional mesh, calculate the difference between the value of a field variable "temperature" at a vertex and the average of field values at its neighboring vertices at time step 13.* Assume "temperature" is the 3rd field variable of a vertex. Note that neighboring vertices are those that share the same parent (i.e., edge) with the given vertex. Then, the query can be written as follows, where the inner select statement retrieves all vertices neighboring a given vertex n.

```
select n.Fields()[3:3] - avg(n2.Fields()[3:3])
  from vertices n,
      (select  n1 from vertices  n1
        where exists (
          select e from elements
           where e.Dimension() = 1 and n1 != n
             and e in n1.Parents()
             and e in n.Parents())) n2
 where timeStep(n) = 13;
```

### 4.2.8 Examples of queries on topology, geometry, and fields combined

We see in Section 3.4 one query type that combines all three categories of mesh operations.

**C-F**: *Given a 2-dimensional mesh, find the total surface area of connected mesh faces at time step 10 such that the value of a field variable "pressure" is higher than 90%.* For a 2-dimensional mesh, a face that is connected to another face is the one that has one or more edges that have faces on both sides. Assuming "pressure" is the 4th field of a mesh face, we can write the query as follows assuming that there exists a function "area" which calculates the geometric area of a mesh face.

```
select sum(area(f)) from elements f
 where f.Dimension() = 2 and timeStep(f) = 10
   and exists (select e from f.Children() e
         where count(e.Parents()) > 1)
           and f.Fields()[4:4] > 90;
```

## 5 Implementation Considerations

We are currently implementing queries on lambda-DB [15], an experimental object-oriented DBMS. It creates mesh elements out of a Silo [3] data file, and visualize query results on MeshTV[4], a visualization tool.

For the rest of this section, we discuss some ideas for efficiently executing queries on the mesh data structure $< \mathcal{G}, \mathcal{N}, \mathcal{F} >$ described in Section 3. First, we can add new classes *3DVolume, Face, Edge,* as well as their associated extents *3dvolumes, faces, edges* to the existing ones. This will enable a query processor to search the concerned extent directly without performing a selection from the extent *elements* based on the attribute "dimension." As a result, a query on mesh elements at a particular dimension, for example faces, can be written as:

```
select f from **faces** f where ...
```

instead of

```
select e from elements e
 where e.Dimension() = 2 and ...
```

An alternative would be to create a hash *index* on the attribute "dimension" of Element extent for a faster retrieval of mesh volumes at a particular dimension.

*Clustering* will obviously improve the performance of certain queries provided that we know what are frequently executed queries. Given our object-oriented model of mesh data, clustering mesh elements is no different than clustering objects in an object-oriented database [13]. One strategy, which will be neutral to different query access patterns, is to cluster all the mesh elements that belong to the same level. Alternatively, clustering can be done by storing each mesh element (e.g., a 3-dimensional volume) together with all its descendent in a given mesh topology graph. Certainly, this brings the issue of handling shared child elements. A conventional resolution is to cluster a child element with one parent and maintain links between the child element and the other parent elements.

As to *indexing* mesh elements, we consider a *two-tier* index on mesh data. As mentioned in Section 4.2.1, we can model a $d$-dimensional mesh volume as a time series of $d$-1-dimensional mesh volumes. So, an index on a time series of mesh data is configured in two tiers – the upper tier on time step, and the lower tier on field variables or geometrical coordinates. Analysis of the queries shown in Sections 4.2.2 through 4.2.8 leads us to create lower tier indexes on the values of the following frequently accessed attributes: attribute “dimension” of class Element for executing the method Dimension(), and attribute “fields” (as a vector or components) of class Element for executing the method Fields(). In addition, retrieving the *first* component of the coordinate of a mesh vertex – used in timeStep(x) of a mesh element x – will benefit from an index on a composite object rooted by x, which is essentially the same as the *nested attribute index* proposed by Bertino and Kim [14]. For example, given a condition “timeStep(v)” where v is a mesh volume, we create an index on mesh volume v for an index key “firstChild(firstChild(firstChild(v))).Coordinate()[0:0]”. One potential problem of a nested attribute index is the complexity of updating the index. However, it is not our concern because simulation mesh data is not subject to update. The data may be entirely replaced, but is not modified.

## 6 Summary and Further Work

The structure and operations of mesh data are far more complex than conventional data used in business applications, such as relational data. Mesh data is used extensively in scientific applications and requires the understanding of mesh topology, mesh geometry, and mesh fields. In this paper, we developed an object-oriented model of mesh data that incorporated all three aspects, proposed a base set of primitive algebraic operators on mesh data, and built an exemplary implementation of the mesh data model in C++. In addition, we demonstrated that the ODMG object query language (OQL) was suitable to express all probable queries we identified through investigation and interaction with scientists. Lastly, we presented our ideas of executing the queries efficiently on the proposed mesh data structure.

In a project [16, 17] that provides a background of this work, an object-relational database management system (ORDBMS) is in use as the platform. Therefore, we desire to implement OQL on an ORDBMS in such a way that an OQL statement is translated to a script of extended SQL statements (e.g, Oracle PL/SQL) and executed on the ORDBMS. This remains as our further work. Another possible future work is to refine the query language based on the sheaf data model [10] mentioned in Section 3, which is based on a solid mathematical theory.

## Acknowledgment

This research was partially supported by University of Vermont UCRS Faculty Research Grant No. PSCI00-1. For LLNL authors, work was performed under the auspices of the U.S. DOE by LLNL under contract No. W-7405-ENG-48.

## References

[1] http://www.unidata.ucar.edu/packages/netcdf/, *UniData NetCDF*.

[2] http://hdf.ncsa.uiuc.edu/, *The NCSA HDF Home Page*.

[3] http://www.llnl.gov/meshtv/manuals.html, *Silo Documentation Version 4.0*.

[4] http://www.llnl.gov/bdiv/meshtv/, *MeshTV: Scientific Visualization and Graphical Analysis Software*.

[5] http://www.atmos.uiuc.edu/envision/envision.html, *ENVISION: an Interactive System for the Management and Visualization of Large Geophysical Data Sets*.

[6] R. Cattell, et al. (ed.), *The Object Data Standard: ODMG 3.0*, Morgan Kaufmann Publishers, January 2000.

[7] W. Wang, J. Yang, and R. Muntz, “STING: A Statistical Information Grid Approach to Spatial Data Mining,” *Proc. Conf. VLDB*, 1997, pp. 86-195.

[8] A. Bauer and W. Lehner, “The Cube-Query-Language (CQL) for Multidimensional Statistical and Scientific Database Systems,” *Proc. Conf. DASFAA*, Melbourne, Australia, April 1-4, 1997, pp.263-272.

[9] A. Shoshani, F. Olken, and H. Wong, “Characteristics of Scientific Databases,” *Proc. Conf. VLDB*, 1993, pp. 147-160.

[10] David M. Butler, “The Sheaf Data Model”, to be published.

[11] D. M. Butler, “A Visualization Model based on the Mathematics of Fiber Bundles,” *Computers in Physics*, September/October 1989, pp. 45-51.

[12] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice in C* (2nd edition), Addison-Wesley, Reading, MA, 1996, pp. 473-476.

[13] P. Valduriez, S. Khoshafian,and G. Copeland, “Implementation Techniques of Complex Objects,” *Proc. Conf. VLDB*, August 1986, pp. 101-109.

[14] E. Bertino and W. Kim, “Indexing Techniques for Queries on Nested Objects,” *IEEE TKDE*, October 1989.

[15] http://lambda.uta.edu/lambda-DB/manual/, L. Fegaras, *lambda-DB*, University of Texas at Arlington.

[16] B. Lee, R. Snapp, and R. Musick, *Ad hoc Query Support for Very Large Scientific Data: the Metadata Approach*, Technical Report UCRL-JC-138481, Lawrence Livermore National Laboratory, Livermore, California.

[17] R. Musick and T. Critchlow, “Practical Lessons in Supporting Large-Scale Computational Science,” *ACM SIGMOD Record*, Vol. 28, No. 4, December 1999.